# Getting started with FRC programming in Java

## Prerequisites

1. Install the development environment
   1. <https://docs.wpilib.org/en/latest/>
2. Install drivers from Cross The Road Electronics
   1. http://www.ctr-electronics.com/hro.html#product\_tabs\_technical\_resources

## Creating the Template

1. Launch FRC VS Code 2020
2. In the upper right corner click the WPIlib Icon
3. Select Create a New Project
4. Select Type
   1. Template
   2. Java
   3. Command Robot
5. Select a folder to save the project
6. Enter a name
7. Team Number : 4632
8. Click “Generate Project”

## Understanding the Template

1. In the file browser expand src
2. Expand java
3. Expand frc
4. Expand robot

### Comments

There are comments in the code that start with // or are multiline comments between the /\*\* and \*// text.

### Main.java

This is the first code that is run when the program starts running. This will call Robot. You will now need to make edits here.

### Robot.java

This defines the Robot object. This is defined code provided to us to execute actions several times a second. These defined code segments are called classes in the Java programming language. The Robot class is provided to us by the WPIlib library.

### RobotContainer.java 2020 format

This defines the RobotContainer object. The Robot class is provided to us by the WPIlib library. As you define components of your robot you will need to add them to this part.

## Including the CTRE library

Click the WPI icon in the upper right

In the box type “Library” and Select “Manage Vendor Libraries”

Select “Install New Libraries (offline)”

Select the checkbox next to “CTRE-Phoenix”

Click OK

## Setting up directory structure

In order to keep things organized you will need to make a few folders. Right click on the folder “robot” and create a New Folder, name it “Subsystems”. Right click on the folder “robot” again and name this one “Commands”.

## Creating a Component

Your robot will be made up of components you will define each component and add it to the robot. Think of this how you would build a robot. The Robot class is the chassis. You will need to add basic components such as drive, control system, and others as that are needed to perform the robot actions.

In order for your code to see the substystems and command you need to import them into your code. Select the Robot.java file and add these lines below the existing import lines.

// Import our commands and subbsystems

import frc.robot.Commands.\*;

import frc.robot.Subsystems.\*;

## Drive Component

For the first component define a drive subsystem. To do this you will need to know the type of motor controller that the robot uses. Team 4632 is using Talon and Victor motor controllers. To create the class do the following steps.

1. Right click on the Subsystems element in the VS Code Explorer
2. Select “Create a new class/command”
3. Select Subsystem
4. Enter “Drive” as the name for the new class

This will create a new class called Drive, this is based on the class Subsystem that is provided by the WPIlib subsystem. You will define the motor controllers and actions that they you will want the drive subsystem to perform. This will be things like moving and stopping.

The package line includes the frc.robot class

The import line include the subsystem class from the WPIlib.

The class line defines the new class.

The template creates a function called initDefaultCommand(), this code will be run as the default for the class. Used when a command is not specified.

The first step you will need to do is to include the libraries add these lines bbel the “Add your docs here comment.

/\*\*

\* These are required libraries to define this drive class

\*/

import com.ctre.phoenix.motorcontrol.can.WPI\_TalonSRX;

import com.ctre.phoenix.motorcontrol.can.WPI\_VictorSPX;

import edu.wpi.first.wpilibj.SpeedControllerGroup;

import edu.wpi.first.wpilibj.drive.DifferentialDrive;

Next add definitions for each motor controller. The variables should have names that represents their function on the root.

private WPI\_TalonSRX leftFront;

private WPI\_VictorSPX leftBack;

private SpeedControllerGroup leftSpeedControllerGroup;

private WPI\_TalonSRX rightFront;

private WPI\_VictorSPX rightBack;

private SpeedControllerGroup rightSpeedControllerGroup;

private DifferentialDrive differentialDrive1;

The WPI\_TalonSRX and WPI\_VictorSPX represent the individual motor controllers. They are then grouped together with SpeedControllerGroup. By using the speed controller group the same commands can be sent to both motors at the same time. The differential drive is the style of drive that will be used. This is a tank style drive.

With the variables defined it is necessary to actually assign values to them. This will be done by creating an initializer function. The initialier function is named the fame as the class. This should be added right below the variable definition

public Drive() {

// Define the left motors

leftFront = new WPI\_TalonSRX(1);

leftBack = new WPI\_VictorSPX(2);

leftSpeedControllerGroup = new SpeedControllerGroup(leftFront, leftBack);

addChild("Left", leftSpeedControllerGroup);

// Define the right motors

rightFront = new WPI\_TalonSRX(3);

rightBack = new WPI\_VictorSPX(4);

rightSpeedControllerGroup = new SpeedControllerGroup(rightFront, rightBack);

addChild("Right", rightSpeedControllerGrooup);

// Define the differential drive will pass the variable for the speed controller groups

differentialDrive1 = new DifferentialDrive(leftSpeedControllerGroup, rightSpeedControllerGroup);

addChild("Differential Drive 1",differentialDrive1);

// Configure parameters for the differential drive

differentialDrive1.setSafetyEnabled(false);

differentialDrive1.setExpiration(0.3);

differentialDrive1.setMaxOutput(1.0);

/\* factory default values \*/

rightFront.configFactoryDefault();

rightBack.configFactoryDefault();

leftFront.configFactoryDefault();

leftBack.configFactoryDefault();

// adjust sensor phase so sensor moves positive when Talon LEDs are green

rightFront.setSensorPhase(true);

leftFront.setSensorPhase(true);

rightBack.setSensorPhase(true);

leftBack.setSensorPhase(true);

}

## Adding functions(methods) to a class

At this point the drive is defined but there are no functions to call to perform specific actions. To start with define a function to drive forward for specific time frame. The DifferentialDrive class provided has a function called arcade. Use the same name for the new function. The parameters are the speed the robot should move values are from -1.0 to 1.0. the value must be specified as a decimal value. The second parameter will be the rotation positive values will be the degrees to turn clockwise. If you hover over the library call in the IDE you will see help information on the function. This is a simple function, but we could add other information like diagnostics or other statements here. The “this” in the call is a reference to the implementation of the class.

public void arcade(double speed, double direction) {

/\* Takes parameters and sets direction \*/

this.differentialDrive1.arcadeDrive(speed, direction);

}

With this the subsystem class you created is defined, but needs to be implemented.

### Create an Instance of the Class

In order to use a class you need to create an instance of it. This is the power of classes they can be used multiple times by declaring multiple instances. To define the instance you will do this in the Robot class. Think if this in the sense that your robot will contain a subsystem. Open the Robobt.java file and find the variable declarations the existing ones will start with private. Add the line below. The “Drive” is the class name and the “driveSubsystem” is the variable name, you can use ny name, but it is important to make it identifiable

public static Drive driveSubsystem;

## Creating a Command

The commands to define actions that you will use repeatedly. There are a few reasons to do this.

1. Commands have functions that are run under certain circumstances, such as when it is finished or interrupted by another action.
2. Created a repeatable action that you will only have to create once and can changes in one location
3. Create actions that act on multiple suybsystems

Right click on “Commands” and select “Create a Command/Subsystem”, when prompted enter the name Forward. This will crate the new class with the required functions of a command class.

## Define dependencies

In the class initializer “Forward” add the following to define required subsystem.

requires(Robot.driveSubsystem);

## Adding the Command to the Robot

Now define the command in the Robot. Open Robot.java and add the Last line in the code segment below. The lines are included for reference on where it goes.

public class Robot extends TimedRobot {

private static final String kDefaultAuto = "Default";

private static final String kCustomAuto = "My Auto";

private String m\_autoSelected;

private final SendableChooser<String> m\_chooser = new SendableChooser<>();

private Forward myAction;

With the variable defined now an instance of the Forward class can be assigned to it. Add the assignment for myAction to robotInit as shown below.

@Override

public void robotInit() {

m\_chooser.setDefaultOption("Default Auto", kDefaultAuto);

m\_chooser.addOption("My Auto", kCustomAuto);

SmartDashboard.putData("Auto choices", m\_chooser);

myAction = new Move();

}

## Adding a Method to the Move class

The Move class now needs a way to actually tell the robot to move. The first method will be a now function to tell the robot to move for a specified duration at a specified speed.

/ Added to initiate the move

public void now(double time, double speed, double direction) {

setTimeout(time);

Robot.driveSubsystem.arcade(speed, direction);

}

## Calling the motion in Autonomous Mode

Update the autonomousInit and add the myAction function call. This is not how you would do it in competition as this will run right away, but will allow for testing.

@Override

public void autonomousInit() {

m\_autoSelected = m\_chooser.getSelected();

System.out.println("Auto selected: " + m\_autoSelected);

myAction(1.0, 0.2, 0.0);

}

This could now be build and deployed for testing. The robot will move at a slow speed for 1 second straight ahead.

# Adding input from a joystick

### Create a OI class

This class will interact with the joystick and the buttons. Create the new class following these instructions.

1. Right click on the Subsystems element in the VS Code Explorer
2. Select “Create a new class/command”
3. Select “Empty Class”
4. Enter “OI” as the name for the new class

### Adding Imports

You will need to import libraries

import edu.wpi.first.wpilibj.Joystick;

### Adding the methods

### Add the methods that will be used to get the forward motion and the directional rotation

public class OI {

Joystick driveStick = new Joystick(0);

// Provide the current X value

public double getRotation() {

double x = this.driveStick.getX();

// Deadband for joystick

if (Math.abs(x) < 0.10) {

x = 0;

}

return x;

}

public double getForwardSpeed() {

double y = this.driveStick.getY();

// Deadband for joystick

if (Math.abs(y) < 0.10) {

y = 0;

}

return y;

}

}

### Add an instance to the Robot

In the Robot.java file add this declaration below the driveSubsystem declaration.

public static OI oi;

Create the instance and assign it, this will be at the end of robotInit.

@Override

public void robotInit() {

m\_chooser.setDefaultOption("Default Auto", kDefaultAuto);

m\_chooser.addOption("My Auto", kCustomAuto);

SmartDashboard.putData("Auto choices", m\_chooser);

driveSubsystem = new Drive();

myAction = new Move();

// Keep this last

oi = new OI();

}

### Create a JoystickDrive class

To create a JoystickDrive class follow the steps below.

1. Right click on the Subsystems element in the VS Code Explorer
2. Select “Create a new class/command”
3. Select Subsystem
4. Enter “JoystickDrive” as the name for the new class

Edit the JoystickDrive and add the requires line as shown below.

public class JoystickDrive extends Command {

public JoystickDrive() {

// Use requires() here to declare subsystem dependencies

// eg. requires(chassis);

requires(Robot.driveSubsystem)

}

Update the execute method to call the drive function.

@Override

protected void execute() {

Robot.driveSubsystem.arcade(Robot.oi.getForwardSpeed(), Robot.oi.getRotation());

}

### Adding the Joystick Drive to the Robot

An instance of JoystickDrive needs to be added to the Robot in order for it to be used. Edit Robot.java and add the line below with the imports.

import edu.wpi.first.wpilibj.command.Command;

Now go to where you added the OI instance and declare the joystickDrive.

Command joystickDrive;

Now assign an instance to the variable. The robotInit should now look like this.

@Override

public void robotInit() {

m\_chooser.setDefaultOption("Default Auto", kDefaultAuto);

m\_chooser.addOption("My Auto", kCustomAuto);

SmartDashboard.putData("Auto choices", m\_chooser);

driveSubsystem = new Drive();

myAction = new Move();

joystickDrive = new JoystickDrive();

oi = new OI();

}